Astronomical Data Analysis Software and Systems VIII
ASP Conference Series, Vol. 172, 1999
D. M. Mehringer, R. L. Plante, and D. A. Roberts, eds.

The STSDAS Table Selector Library

Bernard Simon, Philip Hodge

Space Telescope Science Institute, Baltimore, MD 21218

Abstract. The 2.0.1 release of Tables included the ability to access table subsets. To the user and the programmer such a subset appears to have fewer rows and/or columns than the entire table. The user defines a subset by appending expressions to the table name. This is transparent to the programmer, as the program can just pass the table name to the usual table open routine. Internally, the procedure that evaluates the expressions calls two interpreters: one to evaluate the row selector expression, and another to evaluate the column selector. The syntax for the column selector follows usual Unix pattern matching, and the syntax of the row selector follows the syntax first used in qpoe filters.

1. What Is It?

The library implements a set of procedures that allows the user to select a subset of rows or columns from Stsdas tables. There are two selectors: one for selecting a subset or rows and one for selecting a subset of columns. The information on which rows or columns to select is passed as a string, which is compiled and evaluated by the selector library. The selector library has been embedded in the Stsdas tables library, so that existing tables tasks will allow you to transparently operate on a subset of a table instead of a full table.

Selectors may be applied only to existing tables. They may be used with tables opened read-write, but with some restrictions. When you add rows or columns to a read-write table, for example with tedit, all the new rows and columns are automatically selected. You can add or delete rows at the end of a table that has a row selector in effect, but you cannot add or delete rows before the last row. When you add a row beyond the last selected row, the new row is actually written after the last row in the underlying table. If a row selector is in effect, inserting rows before the end is not allowed because it can be ambiguous as to where they should be inserted. The option of row and column selectors with read-write tables can be very useful with such tasks as tedit or tcalc, but it should be used with caution when reordering the table, such as with tsort.

2. How Can I Use It?

Any program linked with the current version of the tables library can use the column and row selector. Just append the selector expression enclosed in square brackets, with a prefixed "r:" or "c:" to indicate if the row or column selector is being used.

Programmers can also call the selector library directly. The row selector has two interfaces: one which evaluates the expression for a single row and one which evaluates the expression for the entire table. Here is a typical example of the use of the row at a time selector:

```
tp = tbtopn (table, READ_ONLY, NULL)
numrow = tbpsta (tp, TBL_NROWS)
pcode = trsopen (tp, filter)
do irow = 1, numrow {
    if (trseval (tp, irow, pcode)) {
        # Do something neat here
    }
}
call trsclose (pcode)
call tbtclo (tp)
```

The interface which evaluates the expression for an entire table returns a set containing the row numbers for which the expression is true. The set can then be accessed and manipulated by other procedures in the row selector interface. One example of how to use this interface is:

```
set = trsrows (tp, filter)
nset = rst_nelem (set)
do iset = 1, nset {
    irow = rst_rownum (set, iset)
    # do something with the row here
}
call rst_free (set)
```

3. What Is the Syntax?

The column selector filter is a comma separated list of column patterns. A column pattern is either a column name, a file name containing a list of column names, or a pattern using the usual IRAF pattern matching syntax. For example, the string `[c:b,time*,@column.lis]` would be expanded as the column names b, any column name beginning with "time", and all the column names in the file column.lis. If the column list is entirely white space, the array of column descriptors will include all the columns in the table, as this seems the most reasonable default. If the first non-white character is the negation character (!), the array of column descriptors will include all columns not matched by the list. The negation character only has this meaning at the beginning of the list.

Column names may also contain array sections having the same format as image sections. The sections are surrounded by parentheses. For example `[c:spec(1:200:2),image(*,30),spec(20:*)]`. Tasks which use the selector library through the tables library do not support array sections. Array sections are only supported by tasks like tstat, which call the selector library directly.

The row selector uses the same syntax as qpoe filters. For sake of an example, suppose we have a star catalog with the columns Name, Ra, Dec, V, B-V, and U-B. The simplest row selector is an equality test. The name of the column appears on the left of an equals sign and the column value appears on the right. For example, `[r:name=eta_uma]`. Column numbers can be used in place of the column name. This is especially useful for ASCII tables. Values can be either numbers or strings. It is usually not necessary to place strings in quotes. However, if any string (including a column name) contains embedded blanks or characters significant to the qpoe filter, such as equal signs, commas, or colons, it should be placed in quotes.

Ranges of values can be specified by giving the endpoints of the ranges separated by a colon. For example, `[r:v=10:15]` selects all rows with visual magnitude between 10 and 15. Ranges include their endpoints. Ranges can also be used with strings as well as numbers. Ranges can also be one sided. The filter `[r:dec=80:]` selects all rows with declination greater than or equal to eighty degrees and the filter `[r:dec=:-40]` selects all declinations less than or equal to forty degrees south. A filter can contain a list of single values and ranges. The values in the list should be enclosed in parentheses. For example, `[r:name=(eta_uma,alpha_lyr)]` or `[r:b-v=(-1:0,0.5:1)]`.

Individual values or ranges can be negated by placing a ! in front of them. For example, `[r:name=!eta_uma]` selects every row except the star named eta_uma and `[r:ra=!0:6]` selects all rows except those with right ascension between zero and six hours. An entire list can be negated by placing a ! in front of the column name or the parentheses enclosing the list. The filters `[r:!name=(eta_uma,alpha_lyr)]` and `[r:name=!(eta_uma,alpha_lyr)]` are both equivalent.

Filters can test more than one column in a table. The individual tests are separated by commas or semicolons. All tests in the filter must succeed for the filter to be accepted. For example, `[r:ra=1.3:1.4,dec=40:42]` selects a rectangular region in the catalog. A range of row numbers can also be selected by placing the word row on the left side of the equals sign. For example, `[r:row=10:20]` selects rows from ten to twenty inclusive and `[r:row=50:]` selects all rows from fifty on. Row selection can be combined with any other test in a filter. A filter, can also be placed in an include file, for example `[r:@filter.lis]`. Include files can be a part of a larger expression and include files can contain other files, up to seven levels deep.

4. How Does It Work?

The column selector expands a list of column names or patterns into an array of column descriptors. First it checks for a leading negation character, stripping it if found. Then it opens the list of column patterns as a file. It then processes each pattern in the list in a loop. If the pattern is a file name, it pushes the current file descriptor on a stack and opens the new file in its place. If it is a column pattern, it uses the IRAF pattern matching procedures to match the pattern against all column names in the table. All matches are checked against the array of columns already matched and appended if they are not found. If it is a column name, it is checked to see if it is in the table and then added to the array of columns matched if not already in the array. At the end of the file, the previous file descriptor is popped from the stack until no more descriptors are left in the stack. If the pattern was negated, the match array is inverted by comparing it to the columns in the table and including columns not in the match array in the output array. If the pattern is not negated, the match array is the output array.

The tables row selector takes a string which describes the rows to be selected and compiles it into pseudo-code. It then evaluates the pseudo-code to build a data structure containing an array of row numbers. When the task accesses a row, the row number is mapped to the corresponding number in the data structure. Thus to the task the table looks like it only contains the rows selected by the expression.

The row selector expression is generated by a parser generated by yacc. The parser converts the expression into a binary tree. It then checks to see if the expression can be optimized by pre-evaluating row number range subexpressions. If the expression can be optimized, the parser walks the binary tree, evaluating the row number subexpressions and snipping them out of the tree. The modified tree is then converted into postorder pseudo-code. The result of the row number expression is stored as a set of row ranges. If there was no row number subexpression in the original expression, the row number range includes all the rows in the table.

The expression is evaluated for a single row by first checking to see if the row number is in the set of row ranges. If not, the expression is false for that row. If it is, the pseudo-code is evaluated. The evaluator is implemented as a simple select statement in a loop. Values in the table columns are compared to the constants stored in the pseudo-code. The result of the comparison is pushed on a stack. At the end of the evaluation of the pseudo-code, only a single value remains on the stack, which is returned as the result of the evaluation.

Each pseudo-code instruction contains six fields: the operation field, the column field, the true and false jump fields, and the low and high value fields. Not every field is used in each instruction. If a field is not used it is set to zero. The operation field contains an integer code used to choose the branch of the select statement in the evaluator to execute. The column field contains the table column descriptor of the column operated on by the instruction. If set to zero, the row number is used in the operation instead. The jump fields contain the index of the next instruction to execute after the current instruction, depending on the result of the current operation. If the corresponding field is zero, the index is incremented by one instruction instead. The use of jump fields enables the expression evaluator to only compare values to a list until the first success or only check column values until the first failure, speeding up the evaluation of the expression. The value fields contain pointers into a string buffer containing the constant values used by the operation. Although all constants are stored as strings, the evaluator performs numeric as well as string comparisons. The type of comparison is encoded in the operation field.